



ERDC MSRC/PET TR/00-18

A Fortran Interface to POSIX Threads

by

Richard J. Hanson
Clay P. Breshears
Henry A. Gabb

25 May 2000

A Fortran Interface to POSIX Threads

Richard J. Hanson¹

Clay P. Breshears²

Henry A. Gabb³

May 25, 2000

¹Center for High Performance Software Research, Rice University

²Rice University On-site Scalable Parallel Programming Tools Lead for PET

³ERDC MSRC Director of Scientific Computing

**Work funded by the DoD High Performance Computing
Modernization Program ERDC
Major Shared Resource Center through**

Programming Environment and Training (PET)

Supported by Contract Number: DAHC 94-96-C0002
Computer Sciences Corporation

Views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Defense position, policy, or decision unless so designated by other official documentation.

Abstract

Pthreads is the library of POSIX standard functions for concurrent, multithreaded programming. The POSIX standard only defines an application programming interface (API) to the C programming language, not to Fortran. Many scientific and engineering applications are written in Fortran. Also, many of these applications exhibit functional, or task-level, concurrency. They would benefit from multithreading, especially on symmetric multiprocessors (SMP). We present here an interface to that part of the Pthreads library that is compatible with standard Fortran. The contribution consists of two primary source files: a Fortran module and a collection of C wrappers to Pthreads functions. The Fortran module defines the data structures, interface and initialization routines used to manage threads. The stability and portability of the Fortran API to Pthreads is demonstrated using common mathematical computations on three different systems.

1 Introduction

Pthreads is a POSIX standard library [5] for expressing concurrency on single processor computers and symmetric multiprocessors (SMPs). Typical multithreaded applications include operating systems, database search and manipulation, and other transaction-based systems with shared data. These programs are generally coded in C or C++. Hence, the POSIX standard only defines a C interface to Pthreads. The lack of a Fortran interface has limited the use of Pthreads for scientific and numerically intensive applications. However, since many scientific computations contain opportunities for exploiting functional, or task-level concurrency, many Fortran applications will benefit from multithreading.

A thread represents an instruction stream executing within a single address space; multiple threads, of the same process, share this address space. Threads are sometimes called “lightweight” processes because they share many of the properties and attributes of full processes but require minimal system resources to maintain. When an operating system switches context between processes, the entire memory space of the executing process must be saved and the memory space of the process scheduled for execution must be restored. When switching context between threads there is no need to save and restore large portions of memory because the threads are executing within the same memory space. This savings of system resources is a major advantage of using threads.

The Pthreads library provides a means to control the spawning, execution, and termination of multiple threads within a single process. Concurrent tasks are mapped to threads. Threads within the same process have access to their own local, private memory but also share the memory space of the global process. Executing on SMPs, the system may execute threaded tasks in parallel.

As useful as the Pthreads standard is for concurrent programming, a Fortran interface is not defined. The POSIX 1003.9 (FORTRAN Language) committee was tasked with creating “a FORTRAN (77) definition to the base POSIX 1003.1-1990 standard” [3]. There is no evidence of any POSIX standard work to produce a FORTRAN equivalent to the Pthread standard. Fortran 90 has corrected many shortcomings of FORTRAN 77 that may have prevented the formulation of such a standard. There are no serious technical barriers to implementing a workable API in Fortran 90.

We describe the implementation and testing of a Fortran API to Pthreads (FPTHRD). Our tests indicate that the API is standard-complying with Fortran 90 or Fortran 95 compilers. For this reason we use ‘Fortran’ to mean compliance with both standards.

The following section gives some general information on the threaded programming model with specific examples taken from the POSIX library functions. More complete descriptions of the POSIX thread library can be found in [2, 6, 7]. We next give details of the design and implementation of the Fortran API package to the Pthreads library. Section 4 presents threaded example problems and a comparison of their execution performance on three separate SMPs, each with a different native Pthread implementation.

2 Threaded Programming Concepts

Multithreading is a concurrent programming model. Threads may execute concurrently on a uniprocessor system. Parallel execution, however, requires multiple processors sharing the

same memory; i.e., SMP platforms.

Threads perform concurrent execution at the task or function level. A single process composed of independent tasks may break up these computations into a set of concurrently executing threads. Each thread is an instruction stream, with its own stack, sharing the global memory space assigned to the process. Upon completion, the thread's resources are recovered by the system.

All POSIX threads executing within a process are peers. Thus, any thread may cancel any other thread; any thread may wait for the completion of any other thread; and any thread may block any other thread from executing protected code segments. There is no explicit parent-child relationship unless the programmer specifically implements such an association.

With separate threads executing within the same memory address space, there is the potential for memory access conflicts; i.e., write/write and read/write conflicts (also known as race conditions). Write/write conflicts arise when multiple threads attempt to concurrently write to the same memory location; read/write conflicts arise when one thread is reading a memory location while another thread is concurrently writing to that same memory location. Since scheduling of threads is largely non-deterministic, the order of thread operations may differ from one execution to the next. It is the responsibility of the programmer to recognize potential race conditions and control them.

Fortunately, Pthreads provides a mechanism to control access to shared, modifiable data. Locks, in the form of mutual exclusion (mutex) variables, prevent threads from entering critical regions of the program while the lock is held by another thread. Threads attempting to acquire a lock (i.e., enter a protected code region) will wait if another thread is already in the protected region. Threads acquire and release locks using function calls to the Pthreads library.

Pthreads provides an additional form of synchronization through condition variables. Threads may pause execution until they receive a signal from another thread that a particular condition has been met. Waiting and signaling is done using Pthreads function calls.

2.1 POSIX Considerations

The Pthreads header file (**pthread.h**) contains system dependent definitions for data structures that are used with the Pthreads routines. These are typically C structures and are intended to be opaque to the programmer. Manipulation of and access to the contents of the structures should only be done through calls to the appropriate Pthreads functions. Since programmers do not need to deal with differences of structure definitions between platforms, this opacity enables codes to be portable.

Standard names for error codes that can be returned from system calls are established by the POSIX standard. Integer valued constants are defined with these standard names within system header files. As with Pthreads structures, the actual value of any given error code constant may change from one operating system to the next. The names of error codes that may be returned (and the conditions that caused them) from any individual function are listed within the manual pages for the function. As with Pthreads structures, the intention is to keep the specific values given to each error code hidden from the programmer. Thus,

the programmer need only compare a function's return value against the named constant to determine if a specific error condition has arisen.

3 Design and Implementation of Fortran API

The Pthreads library is relatively small, consisting of only 61 routines that can loosely be classified into three categories: thread management, thread synchronization, and thread attributes control. Thread management functions deal with the creation, termination, and other manipulation of threads. The two methods available for guaranteeing the correct and synchronous execution of concurrent threads are mutex and condition variables. These constructs, and the functions to handle them, are used to ensure the integrity of data being shared by threads. The Pthreads standard defines attributes in order to control the execution characteristics of threads. Such attributes include detach state, stack address, stack size, scheduling policies, and execution priorities.

3.1 Fortran Interface Details

The FPTHRD package consists of a Fortran module and file of C routines. The module defines Fortran derived types, parameters, interfaces, and routines to allow Fortran programmers to use Pthread routines. The C functions provide the interface from Fortran subroutine calls and map parameters into the corresponding POSIX routines and function arguments.

The following sections describe some of the design decisions we faced and the similarities and differences between FPTHRD and the Pthreads standard.

3.1.1 Naming Conventions

The names of the FPTHRD routines are derived from the Pthreads root names; i.e., the string following the prefix **pthread_**. The string **fpthrd_** replaces this prefix. In this way, a call to the Pthreads function **pthread_create()** translates to a call to the Fortran subroutine **fpthrd_create()**. Our initial thoughts were to prefix the full POSIX names with the character **f**, which would yield the prefix string **fpthread_** before each root name. However, the Fortran standard [1] limits subroutine and variable names to 31 characters. The longest POSIX defined name is 32 characters in length. Since the **fpthrd_** prefix yields a net loss of one character over the POSIX prefix, we can guarantee that routine names in our package will have no more than 31 characters. All the Fortran routine names are therefore standard-compliant and all the Pthreads root names remain intact.

For consistency, all POSIX data types (Table 1) and defined constants (Table 2) prefixed with **pthread_** (**PTHREAD_**) are defined with the prefix **fpthrd_** (**FPTHRD_**) within the Fortran module. Besides those defined specifically for Pthreads types, other POSIX types are used as parameters to Pthreads functions. For these additional structures a corresponding definition is included within the module with the prefix character **f** added to the POSIX name.

Table 1 Correspondence Between Pthreads Types and FPTHRDL Derived Types

POSIX Structure Name	Fortran Derived Type Name
pthread_t	TYPE (fpthrd_t)
pthread_once_t	TYPE (fpthrd_once_t)
pthread_attr_t	TYPE (fpthrd_attr_t)
pthread_mutex_t	TYPE (fpthrd_mutex_t)
pthread_mutexattr_t	TYPE (fpthrd_mutexattr_t)
pthread_cond_t	TYPE (fpthrd_cond_t)
pthread_condattr_t	TYPE (fpthrd_condattr_t)
sched_param	TYPE (fsched_param)
timespec	TYPE (ftimespec)
size_t	TYPE (fsize_t)

Table 2 Correspondence Between Pthreads Constants and FPTHRDL Equivalents

POSIX Constant Name	FPTHRDL Parameter
PTHREAD_CREATE_DETACHED	FPTHRDL_CREATE_DETACHED
PTHREAD_CREATE_JOINABLE	FPTHRDL_CREATE_JOINABLE
PTHREAD_PROCESS_PRIVATE	FPTHRDL_PROCESS_PRIVATE
PTHREAD_PROCESS_SHARED	FPTHRDL_PROCESS_SHARED
PTHREAD_PRIO_PROTECT	FPTHRDL_PRIO_PROTECT
PTHREAD_PRIO_INHERIT	FPTHRDL_PRIO_INHERIT
PTHREAD_PRIO_NONE	FPTHRDL_PRIO_NONE
PTHREAD_CANCEL_ENABLE	FPTHRDL_CANCEL_ENABLE
PTHREAD_CANCEL_DISABLE	FPTHRDL_CANCEL_DISABLE
PTHREAD_CANCEL_DEFERRED	FPTHRDL_CANCEL_DEFERRED
PTHREAD_CANCEL_ASYNCHRONOUS	FPTHRDL_CANCEL_ASYNCHRONOUS
PTHREAD_CANCELED	FPTHRDL_CANCELED
PTHREAD_SCOPE_SYSTEM	FPTHRDL_SCOPE_SYSTEM
PTHREAD_SCOPE_PROCESS	FPTHRDL_SCOPE_PROCESS
PTHREAD_INHERIT_SCHED	FPTHRDL_INHERIT_SCHED
PTHREAD_EXPLICIT_SCHED	FPTHRDL_EXPLICIT_SCHED
PTHREAD_STACK_MIN	FPTHRDL_STACK_MIN
PTHREAD_THREADS_MAX	FPTHRDL_THREADS_MAX
SCHED_RR	SCHED_RR
SCHED_FIFO	SCHED_FIFO
SCHED_OTHER	SCHED_OTHER

Figure 1 Example Code – Use of Static Initializers

```

TYPE (fpthrd_mutex_t) :: any_mutex
:
any_mutex = FPTHRD_MUTEX_INITIALIZER

```

3.1.2 Structure Initialization

Besides the routines specifically designed for initialization, the Pthreads library includes predefined constants that can be used to initialize mutexes, condition variables, and ‘once block’ structures to their default values. Corresponding derived type constants for initialization have been defined and included in FPTHRD. The type and names of these initialization constants for a condition variable, mutex variable, and ‘once block’ variable are:

```

TYPE (fpthrd_cond_t) :: FPTHRD_COND_INITIALIZER
TYPE (fpthrd_mutex_t) :: FPTHRD_MUTEX_INITIALIZER
TYPE (fpthrd_once_t) :: FPTHRD_ONCE_INIT

```

To use these initialized data types with default attributes, assign the value in a program unit with the assignment operator (Figure 1).

3.1.3 Parameters

The Fortran API preserves the order of the arguments of the C functions and provides the C function value as the final argument. This style of using Fortran subroutines for corresponding C functions with the return argument appended to the parameter list is used in the Fortran API for both MPI [8] and PVM [4]. This trailing integer argument is most often used to return an indicator of the termination status of the routine. A return value of zero indicates that the routine call did not yield any exception; any non-zero return value indicates that an exception condition occurred during execution. Whether an exception condition is an error or can be ignored is determined in the context of the application. The POSIX standard defines names for specific conditions and requires fixed integer values be attached to these error codes. The Fortran module defines integer constants with the same names as the POSIX standard for all potential error codes that might be returned from Pthreads functions. The values of these Fortran constants are the same as their POSIX counterparts on the target platform. The routines **fpthrd_self()** and **fpthrd_equal()** have no status argument since they do not return exception flags.

Fortran provides compile-time checking of argument type, number, kind, and rank using interface blocks. This is an advantage over the C programming language, which does not provide argument checking. Besides the compile-time checking, interface blocks also provide for argument overloading. This feature allows the use of **TYPE(C_NULL)** parameters where an optional NULL could be used in the underlying C functions. Fortran interface blocks also make it possible for the status parameter to be optional in Fortran routine calls. The module in our package provides interface blocks for the Fortran routines that call corresponding C functions with the exception of routine **fpthrd_create()**. Since the argument type for

the threaded subroutine is chosen by the program author, it is necessary to exclude type checking for **fpthrd_create()**. The status parameter is not optional in calls to this routine.

3.1.4 **fpthrd_join()** Parameters

One special case should be mentioned with respect to parameters. The second parameter of **fpthrd_join()** is used to return an exit code from the **fpthrd_exit()** call of the thread being joined. The Pthreads library uses a **void **** type to allow the return of any defined data value or struct. If no value is expected or needed by the joining thread, a NULL value may be used.

Due to the difference in the way C and Fortran pointers are implemented (see §3.3 for further discussion) and the desire to keep the programming of the interface as simple as possible, it was decided to restrict the type of this parameter to **INTEGER**. This type restriction is repeated in the single parameter of the **fpthrd_exit()** routine which generates the value.

Within scientific applications, it was thought that this exit value would be used mostly for returning a completion code to the joining thread. Special codes could be designed to signify the success or failure, and the cause of any failure, of the joined thread. Should more elaborate data structures be required to be passed from a thread to that thread which joins it, the integer value can be used as a unique index into a global array of results.

3.1.5 Description of Routines and Arguments

Appendix A contains a brief description of the functionality of each routine included in the Fortran API along with the types and order of the parameters. We refer the reader to [2, 6, 7] for a more complete description of Pthreads functions.

3.2 Support and Utility Routines

This section contains details on several routines that are not included in the Pthreads standard. These routines have been included in FPTHR to provide the programmer the ability to give the runtime system a hint as to the number of active threads desired, to initialize the Fortran API routines and check parameter values and derived type sizes, and to manipulate POSIX defined data types required by certain Pthreads functions for which there is no Fortran compliant method generally available. Details on the parameter types and order for the routines described here are included in Appendix B.

Many systems that support multithreading have an included function to inform the runtime system of the number of threads the system should execute concurrently. This seems to be particularly relevant for uniprocessor systems and is intended to allow finer control of system resources by the thread programmer. We have included the **fpthrd_setconcurrency()** and **fpthrd_getconcurrency()** subroutines in FPTHR in order to give the programmer the chance to request the number of kernel entities that should be devoted to threads; i.e., the number of threads to be executing concurrently. If the target platform does not support this functionality, calls to these routines will return without altering anything.

An initial data exchange is required as a first program step before using other routines in FPTHR. Initialization is performed with a call to the routine **fpthrd_data_exchange()**.

Figure 2 Example Code – Use of Error Codes

```

DO
  CALL fpthrd_create(tid, NULL, thred_routine, routine_arg, ierr)
  IF (ierr /= EAGAIN) EXIT
  CALL wait_some_random_time
END DO

```

This routine is similar in functionality to the **MPL_INIT()** routine from MPI. The data exchange was found to be necessary because the parameters defined in Fortran or constants defined in C are not directly accessible in the alternate language. One such value of note is the parameter **NULL** passed from Fortran to C routines. This integer is used as a signal within the C wrapper code to substitute a **NULL** pointer for the corresponding function argument. The derived type **TYPE(C_NULL)**, while available to programmers, is not meant for use except to define the special parameter value **NULL**.

The working space for the C structures of Pthreads data types is stored as Fortran derived types. Each of the definitions for derived types is an integer array with the **PRIVATE** attribute. Pthreads structures are opaque. The **PRIVATE** attribute prevents the Fortran program from inadvertently accessing the data in these structures. One other task performed by the **fpthrd_data_exchange()** routine is to ensure that the Fortran derived types are of sufficient size to hold the corresponding C structures.

Five additional routines are included to give the programmer the ability to manipulate those C structures used by Pthreads that are not a direct part of the Pthreads definition. The Fortran names defined in **FPTHRD** for these data types are **TYPE(fsize_t)**, **TYPE(ftimespec)**, and **TYPE(fsched_param)** (as shown in Table 1). For these data types there are routines to set and retrieve values from objects of each type.

3.2.1 Error Checking

The POSIX standard defines a set of error codes that may be returned from calls to Pthreads functions that signal when exceptional conditions have occurred. These exception codes are available from the routines in **FPTHRD** through the optional status parameter. Examination of the returned value of the status parameter allows codes to dynamically react to possibly fatal conditions that may arise during execution.

As an example, consider a code that requires the creation of a large number of threads. During execution, resources may be temporarily unavailable to create new threads. Rather than abort the entire computation at this exception, it would be prudent to pause the creation of new threads until resources become available. In the event that the **fpthrd_create()** status parameter return value be equal to the **EAGAIN** error constant, the spawning thread would wait for some amount of time before attempting to create another thread (Figure 2). As long as the **EAGAIN** exception value is returned from **fpthrd_create()**, the spawning thread will continue to wait before attempting to create the new thread.

While each platform may have different values for **EAGAIN** and all other error constants, the initial data exchange routine accounts for these differences. All the programmer needs

Figure 3 Example Code – Use of **ferr_abort()**

```

    print *, 'Join the same thread twice.'
    call FPTHRD_join(thread, NULL, status)
    call ferr_abort (5, status, ", joining thread")
    print *, 'First JOIN okay.'
    call FPTHRD_join(thread, NULL, status)
    call ferr_abort (6, status, ", joining thread")
    print *, 'Second JOIN okay.'
```

to do is use the symbolic name; e.g., **EAGAIN**. The possible error constants that may be returned from each routine in **FPTHRD** are detailed in the ‘man’ page for each routine.

Since most routines in **FPTHRD** have several possible exception codes, rather than specifically check for each one, a method to print out what exception code was returned may be desired. This is especially true when debugging threaded applications. **FPTHRD** contains a routine, **ferr_abort()**, that provides the functionality described above as well as aborting further processing by all threads. A brief description of the **ferr_abort()** subroutine and its parameters is given below.

```

SUBROUTINE ferr_abort(sequence_number, status, text_string)
    INTEGER, INTENT(IN) :: sequence_number
    INTEGER, INTENT(IN) :: status
    CHARACTER, DIMENSION(*), INTENT(IN) :: text_string
```

The sequence number is an arbitrary identifying integer printed with the error message. The status argument is a variable holding the exception code value returned from a prior call to some routine in **FPTHRD**. If the status value is non-zero, a message containing the corresponding error constant is printed along with the text of the third argument. A Fortran **STOP 'Abort'** is also executed to terminate the computation. If the status value is zero, no action is taken by the **ferr_abort()** routine. Thus, it is safe (and very wise) to insert calls to **ferr_abort()** after calls to **FPTHRD** routines when fatal errors are possible. Where it is possible that non-fatal exceptions may be encountered, these should be dealt with directly by the application code.

As an example, a call to **fpthrd_join()** to a given thread after that thread has already been joined yields an exception (Figure 3). The output (Figure 4) indicates the thread does not exist because it was joined in a previous call. (This would often be considered a non-fatal exception, but one that the programmer would wish to know had occurred.) The next-to-last line in the message beginning with “**Unix error ...**” contains verbatim text returned from the Unix character utility, **strerror(status)**.

Figure 4 Example Output – Use of `ferr_abort()`

```

Join the same thread twice.
First JOIN okay.
Failed 6 with value 3, joining thread
Error codes are valid after statement: CALL FPTHRD_DATA_EXCHANGE()
      3  ESRCH          No such thread exists.
Unix error summary:  No such process
STOP Abort

```

3.3 What’s Not Included in the Package

The functionality of several routines included in the Pthreads library is outside the scope of Fortran. We describe these functions in this section and state our reasons for their exclusion from FPTHRD.

The functions `pthread_cleanup_push()` and `pthread_cleanup_pop()` allow the programmer to place and remove function calls into a stack structure. Should a thread be canceled before the corresponding `pop` calls have been executed, the functions are removed from the stack and executed. In this way, threads are able to “cleanup” details such as allocated memory even if normal termination is thwarted.

While the functionality of these routines is desirable, they are typically implemented as macros defined in the `pthread.h` header file in order to ensure paired `push` and `pop` calls. Upon further examination, we have found undocumented system calls and data structures used within these macros. Since the targets for FPTHRD are scientific computation and numerical codes, it was concluded that such functionality might not be as useful as other functions. With that in mind, it was decided the effort required to develop a simple, general algorithm to implement equivalent cleanup functions in Fortran outweighed the potential benefit.

In order to understand the problems inherent in the functions associated with thread-specific data—`pthread_getspecific()`, `pthread_key_create()`, `pthread_key_delete()`, and `pthread_setspecific()`—we examine the function `pthread_getspecific()`. This function returns a `void` C pointer to a data object associated with the calling thread. This allows local data pertinent to a user’s threaded function to be available before the thread terminates. Fortran defines a pointer attribute for intrinsic and derived types (from [1]):

“...a pointer is a descriptor with space to contain information about the type, type parameters, rank, extents, and location of a target. Thus a pointer to a scalar object of type `real` would be quite different from a pointer to an array of user-defined type. In fact each of these pointers is considered to occupy a different unspecified storage unit.”

A C pointer is simply a memory address. As evidenced from the above passage, Fortran cannot access or manipulate memory addresses directly. At this time, we can find no portable way to implement the thread-specific data functions without imposing obstructive constraints.

The functions `pthread_attr_getstackaddr()` and `pthread_attr_setstackaddr()` manipulate a thread's stack address. As stated previously, Fortran has no facility to directly manipulate memory addresses. Besides, implementation details, such as a stack, are not addressed in the Fortran standard. Thus, there is a danger that a Fortran program that calls these routines may not recognize setting of the stack address.

The `pthread_atfork()`, `pthread_kill()`, and `pthread_sigmask()` functions deal with the `fork()` function and interthread signaling. Since support for these features from Fortran programs within the runtime system is unknown and perhaps even unsupported, especially between different operating systems, these functions are not included in FPTHR. D.

3.4 Package Contents

FPTHR. D consists of a Fortran module, `fpthrd.f90`, and a file of C functions, `ptf90.c`, and an include file, `summary.h`. We also have included four test/verification programs, timing programs for matrix-vector product and matrix transpose, 'man' pages for each function within FPTHR. D, and other documentation.

4 Performance Benchmarks and Test Results

A primary reason for a programmer to use thread technology is to enhance performance of an application code. In this section we have included timing information that demonstrates threaded efficiency for computing a matrix-vector product and transposing storage of an array. The size of the square matrices is $n = 1023$ rows and columns. Small values of n are not likely to see benefit due to the system overhead of managing the threads.

4.1 Matrix-Vector Product

For this benchmark, the task is to compute $y = Ax$, where A is an $n \times n$ real matrix and x and y are both real vectors of length n . To this end consider the compatible partitioning $A = [A_1|A_2|\cdots|A_k]$ and $x = [x_1|x_2|\cdots|x_k]^T$ where each A_i is a consecutive set of columns from A and each x_i is a subvector of x that contains the corresponding elements as the columns in A_i . The partitioning of A provides the computation $y = \sum_{i=1}^k A_i x_i$. Each term of the sum may be computed concurrently. We examine three methods for accomplishing this computation: simple loops, the Fortran `MATMUL` intrinsic, and threaded.

4.1.1 Simple Loops

This method uses an ordinary doubly nested loop (Figure 5). The inner loop moves with unit strides in accessing the entries of the arrays `Y(:)` and `A(:, :)`. (We realize that `X(:)` is also accessed with unit stride, but for each iteration of the I-loop, `X(J)` is a constant.) Programmers are likely to use this algorithm because it is simple and gives good performance for small values of n . Potential refinements of this computation include unrolling the loops, which may improve performance due to more efficient use of the instruction and data cache.

Figure 5 Example Code – In-Line Product

```

Y = 0.0
DO J = 1, N
  DO I = 1, N
    Y(I) = Y(I)+A(I,J)*X(J)
  END DO
END DO

```

Figure 6 Example Code – MATMUL Intrinsic Function

```

Y=MATMUL(A,X)

```

4.1.2 MATMUL Intrinsic

A programmer can use the Fortran **MATMUL** intrinsic function for computing the matrix-vector product. To use the intrinsic function the programmer writes a single line (Figure 6).

4.1.3 Threaded Products

We concurrently compute the terms $t_i = A_i x_i$ with a thread-safe subprogram. A mutex (P) is used to coordinate the accumulation of the sum, $y \leftarrow y + t_i$. The partitioning of the problem is done using recursion to build a family of threads that ultimately computes a partial sum by calling the basic subprogram using the simple loops algorithm given in §4.1.1.

When the number of columns in a thread's current assigned submatrix, A_j , reaches a defined ideal value, the simple loops algorithm is called to compute the partial sum using A_j and the corresponding x_j . This partial sum is then added to the global sum. Alternatively, two recursive threads are created and later joined. Each thread is assigned a subproblem half the size of the problem assigned to the spawning thread. That is, consecutive columns of A_j are divided into two approximately equal-sized submatrices and each portion is assigned to one of the new threads.

While the basic matrix-vector subprogram may use loop unrolling to gain additional performance, the biggest gain from threading is being able to take advantage of the inherent concurrency of the approach and the fact that there are multiple threads used for the computations.

Figures 7, 8, and 9 contain a Fortran module, support subroutines, and main program that implement this recursive algorithm.

We do not claim that our threaded algorithm is optimal, only that it is correct and significantly improves performance over the other standard approaches on the hardware platforms tested. The threshold used to determine the ideal size of submatrices for simple loop execution on all machines was set to generate eight threads that performed this computation. This eight-thread limit was set because the SUN Enterprise system that was available had eight processors, each with a separate instruction and data cache. Also, the IBM Power3 SMP platform used was configured with eight processors per SMP node.

Figure 7 Example Code – Module for Threaded Matrix-Vector Product

```

MODULE MODULE_THREADED_PRODUCT
  USE FPTHREAD
  IMPLICIT NONE

  ! This is the maximum number of threads that do actual computation.
  INTEGER, PARAMETER :: NTHREADS = 8
  INTEGER, PARAMETER :: NSIZE = 1023

  TYPE(FPTHREAD_MUTEX_T) :: P_MUTEX

  ! Define a derived type that will carry the problem information.
  TYPE FUNCTION_ARGUMENTS
    INTEGER :: N
    REAL, DIMENSION(:,:), POINTER :: SMATRIX
    REAL, DIMENSION(:), POINTER :: SX
    REAL, DIMENSION(:), POINTER :: SY
  END TYPE

  TYPE(FUNCTION_ARGUMENTS) :: INA

  ! Define the interface to the routine called by the threaded routine.

  INTERFACE
    SUBROUTINE FSGEMV (N, SMATRIX, SX, SY, ROW_START, ROW_END)
      IMPLICIT NONE
      INTEGER :: N, ROW_START, ROW_END
      REAL, POINTER, DIMENSION(:) :: SMATRIX(:, :), SX, SY
    END SUBROUTINE FSGEMV

    RECURSIVE SUBROUTINE THREADING_PRODUCT(LIMITS)
      INTEGER :: LIMITS(2)
    END SUBROUTINE THREADING_PRODUCT
  END INTERFACE

END MODULE MODULE_THREADED_PRODUCT

```

Figure 8 Example Code – Routines for Threaded Matrix-Vector Product

```

RECURSIVE SUBROUTINE THREADING_PRODUCT(LIMITS)
  USE MODULE_THREADED_PRODUCT, STARTED_ROUTINE=>THREADING_PRODUCT
  ! Symbol is reset to a dummy. This avoids error messages in some compilers.
  IMPLICIT NONE

  INTEGER :: IDEAL, K, J, LIMITS(2), LIMITS_L(2), LIMITS_R(2), STATUS
  TYPE(FPTHRD_T) :: THREAD_L, THREAD_R

  K = LIMITS(2)-LIMITS(1)+1;  J = (LIMITS(1)+LIMITS(2))/2

  ! The problem LIMITS are split into two equally sized groups.
  LIMITS_L = LIMITS;  LIMITS_R = LIMITS
  LIMITS_L(2) = J;  LIMITS_R(1) = J+1

  IDEAL = (INA%N+NTHREADS-1)/NTHREADS

  IF (K <= IDEAL) THEN
    ! This routine is where the work actually gets done.
    ! The above value of IDEAL is used to create about NTHREADS threads
    ! across the multi-generational family.
    CALL FSGEMV (INA%N, INA%SMATRIX, INA%SX, INA%SY, LIMITS(1), LIMITS(2))
  ELSE

    ! Create a new family of two threads and reduce the problem size
    CALL FPTHRD_CREATE(THREAD_L, NULL, STARTED_ROUTINE, LIMITS_L, STATUS)
    CALL FERR_ABORT(5, STATUS, " recursive create-L in THREADING_PRODUCT")
    CALL FPTHRD_CREATE(THREAD_R, NULL, STARTED_ROUTINE, LIMITS_R, STATUS)
    CALL FERR_ABORT(6, STATUS, " recursive create-R in THREADING_PRODUCT")

    CALL FPTHRD_JOIN(THREAD_L, NULL, STATUS)
    CALL FERR_ABORT(7, STATUS, " recursive join-L in THREADING_PRODUCT")
    CALL FPTHRD_JOIN(THREAD_R, NULL, STATUS)
    CALL FERR_ABORT(8, STATUS, " recursive join-R in THREADING_PRODUCT")
  END IF
END SUBROUTINE THREADING_PRODUCT

```

Figure 8 (Continued) Example Code – Routines for Threaded Matrix-Vector Product

```
SUBROUTINE FSGEMV (N, SMATRIX, SX, SY, COL_START, COL_END)
  USE MODULE_THREADED_PRODUCT, NOT_USED=>FSGEMV
  IMPLICIT NONE
  INTEGER :: N, I, J, COL_START, COL_END, STATUS
  REAL, POINTER :: SMATRIX(:, :), SX(:)
  REAL, POINTER :: SY(:)
  REAL :: T(N)

  ! Compute a partial sum of the matrix-vector product.
  T = 0.0
  DO J = COL_START, COL_END
    DO I = 1, N
      T(I) = T(I)+SMATRIX(I,J)*SX(J)
    END DO
  END DO

  ! Update the global vector sum, eventually with all partial sums.
  ! A lock is required to prevent a 'race' condition.
  CALL FPTHRD_MUTEX_LOCK(P_MUTEX, STATUS)
  DO I = 1, N
    SY(I) = SY(I)+T(I)
  END DO
  CALL FPTHRD_MUTEX_UNLOCK(P_MUTEX, STATUS)
END SUBROUTINE FSGEMV
```

Figure 9 Example Code – Main Program Computing Matrix-Vector Product

```

PROGRAM MATRIX_VECTOR_PRODUCT
  USE MODULE_THREADED_PRODUCT
  IMPLICIT NONE

  REAL, POINTER :: MATRIX_A(:, :), VECTOR(:), Y_SERIAL(:), Y_THREAD(:)
  REAL :: ERRNORM, NORM, TEMP
  INTEGER :: I, J, N, STATUS, LIMITS(2)
  TYPE(FPTHRD_T) :: THREAD_ID

  CALL FPTHRD_DATA_EXCHANGE()
  CALL FPTHRD_SETCONCURRENCY(NTHREADS+1, STATUS)
  CALL FPTHRD_MUTEX_INIT(P_MUTEX, NULL, STATUS)
  CALL FERR_ABORT (1, STATUS, " initializing P_MUTEX")

  ! The Fortran random number generator is used to generate
  ! an arbitrary sequence of matrix and vector values.
  N = NSIZE
  ALLOCATE(MATRIX_A(N,N), VECTOR(N), Y_SERIAL(N), Y_THREAD(N))
  CALL RANDOM_NUMBER(MATRIX_A)
  CALL RANDOM_NUMBER(VECTOR)

  ! Compute the matrix-vector product for comparison.
  Y_SERIAL = 0.0
  DO J = 1, N
    DO I = 1, N
      Y_SERIAL(I) = Y_SERIAL(I)+MATRIX_A(I,J)*VECTOR(J)
    END DO
  END DO

  ! Define contents of the ad-hoc derived type for the problem data.
  ! This is the size and array definitions needed by the threads.
  INA%N = N; INA%SMATRIX=>MATRIX_A; INA%SX=>VECTOR; INA%SY=>Y_THREAD

  ! Create a single thread for computing the product. Start this thread
  ! with the entire problem size as the argument to the recursive routine.
  LIMITS = (/1,N/)
  Y_THREAD = 0.0
  CALL FPTHRD_CREATE(THREAD_ID, NULL, THREADING_PRODUCT, LIMITS, STATUS)
  CALL FERR_ABORT(1, STATUS, " creating a recursive thread")

```

Figure 9 (Continued) Example Code – Main Program Computing Matrix-Vector Product

```

    CALL FPTHRD_JOIN(THREAD_ID, NULL, STATUS)
    CALL FERR_ABORT(1, STATUS, " joining a single recursive thread")

! Check results for correctness:
    ERRNORM = SUM((Y_SERIAL-Y_THREAD)**2)
    NORM = SUM(Y_SERIAL**2)

! The results are correct even if they do not completely agree.
! This test will fail for large errors. Small errors will be tolerated.

    IF(ERRNORM > EPSILON(NORM)*NORM) THEN
        PRINT *, "ERROR! Serial and threaded algorithms gave different results!"
    END IF

END PROGRAM MATRIX_VECTOR_PRODUCT

```

Figure 10 Example Code – Simple Loops

```

    DO J = 1, N
        DO I = 1, N
            B(J,I) = A(I,J)
        END DO
    END DO

```

4.2 Matrix Transposition

For this benchmark, the task is to assign the array $B = A^T$, where A and B are both $n \times n$ real matrices. Using the partitioning scheme from the previous benchmark, we can describe the transpose as $B = [A_1|A_2|\cdots|A_k]^T$. Each group of rows for B corresponds to a set of columns for A , and the storage process of each group may be concurrent. We examine three methods for accomplishing this computation: simple loops, the Fortran `TRANSPOSE` intrinsic, and threaded.

4.2.1 Simple Loops

This method uses an ordinary doubly nested loop (Figure 10). The inner loop moves with unit strides in accessing the entries of the arrays $A(:, :)$. Fortran uses column-major storage so the stride for access to elements of $B(:, :)$ is n . Programmers are likely to use this loop because it results in good performance for small values of n . Refinements of this computation include unrolling the loops.

Figure 11 Example Code – TRANSPOSE Array Intrinsic

```
B = TRANSPOSE(A)
```

Table 3 Speedup Ratios for Matrix-Vector Products Codes

Machine Details	Simple Loop vs Threaded Product	MATMUL() vs Threaded Product
SUN Enterprise 4000	5.82	1.46
SGI Origin 2000	2.72	1.80
IBM Power3 SMP	3.20	3.30

4.2.2 TRANSPOSE Intrinsic

A programmer can use the Fortran TRANSPOSE intrinsic function for computing the matrix transposition. To use the intrinsic function the programmer writes a single line (Figure 11).

4.2.3 Threaded Transpose

For this version of the code, data is concurrently moved from block columns of array **A** to the block rows of array **B**. The partitioning of the problem is done recursively. When the recursion is halted, the code completes the assignment using the simple code described in §4.2.1. The code for this application is similar to that for the matrix-vector products. Thus, we do not include the listings.

4.3 Timing Results

The data shown in Tables 3 and 4 are the speedup execution time for the in-line and intrinsic function implementations compared to the threaded version. The simple loop matrix-vector product code ran 5.82 times longer than the threaded code on the SUN Enterprise 4000 system; the simple loop transpose code ran 9.03 times longer than the threaded code on the IBM Power3 SMP system. We avoided giving absolute times due to the fact that the machines we used had varying computational power, and this could lead to the erroneous conclusion that one machine was superior to the other two. The speedups demonstrate that the use of threads enhances performance on all of these machines.

The test codes each timed repetition of the computations 16 times within a single run. Each of these runs was repeated 32 separate times, and a median time computed for each code. These median values were used to compute the speedup ratios. Because of the variations in system load over time, it was decided that the use of median time across repeated executions would yield a fairer value for comparison between each code. All timing runs were executed on each platform using eight processors. The highest level of optimization available in the Fortran compiler was used on each code.

Table 4 Speedup Ratios for Array Transpose Codes

Machine Details	In-Line $B = A^T$ vs Threaded Transpose	TRANSPOSE() vs Threaded Transpose
SUN Enterprise 4000	4.15	1.43
SGI Origin 2000	2.54	4.32
IBM Power3 SMP	9.03	5.01

5 Summary

In order for scientific and numerical codes to take full advantage of new SMP architectures, some shared-memory programming model must be used. Pthreads is one such model for shared-memory, concurrent programming. However, the primary language of scientific and numerical computing is Fortran and a standard Pthreads interface is only defined for the C language.

We have described the design and implementation of a Fortran API (FPTHRD) to the Pthreads library. Our package allows Fortran programmers to harness the computing power offered by Pthreads without having to program in C or to develop their own interlanguage interface. We used FPTHRD within two benchmarking codes in order to demonstrate the efficacy of threaded programming for scientific computation. FPTHRD was tested on three operating systems to demonstrate portability.

Acknowledgments

We express our gratitude to Dr. Rob Fowler (Rice University) and Joseph Robichaux (IBM) for advice and guidance during the development of this package. We also thank Mr. John Bachir (Rice University) for his help testing the software.

References

- [1] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. *Fortran 95 Handbook*. The MIT Press, Cambridge, MA, 1997.
- [2] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, Reading, MA, 1997.
- [3] David R. Butenhof. Personal communication, 1999.
- [4] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: A Users' Guide and Tutorial for Network Parallel Computing*. The MIT Press, Cambridge, MA, 1994.
- [5] 9945-1:1996 (ISO/IEC) [IEEE/ANSI Std 1003.1 1996 Edition] *Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application: Program Interface (API) [C Language] (ANSI)*, IEEE Standards Press, 1996.

- [6] Bil Lewis and Daniel J. Berg. *Multithreaded Programming with Pthreads*. Sun Microsystems Press, Mountain View, CA, 1998.
- [7] Bradford Nichols, Dick Buttlar, and Jacqueline Prolux Farrell. *Pthreads Programming*. O'Reilly and Associates, Sebastopol, CA, 1996.
- [8] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI—The Complete Reference: Volume 1, the MPI Core*. The MIT Press, Cambridge, MA, 1998.

APPENDIX A – Fortran Interface to Pthreads Functions

This appendix contains information detailing the types and order of the parameters of the Pthreads functions included in FPTHRD as well as a brief description of the functionality of each routine.

```
SUBROUTINE fpthrd_attr_destroy(attr, status)
```

```
TYPE (fpthrd_attr_t), INTENT(OUT) :: attr
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Destroys the thread attribute object.

```
SUBROUTINE fpthrd_attr_getdetachstate(attr, createstate, status)
```

```
TYPE (fpthrd_attr_t), INTENT(IN)  :: attr
```

```
    INTEGER, INTENT(OUT) :: createstate
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Returns current setting of detachstate defined in thread attribute.

```
SUBROUTINE fpthrd_attr_getinheritsched(attr, inheritsched, status)
```

```
TYPE (fpthrd_attr_t), INTENT(IN)  :: attr
```

```
    INTEGER, INTENT(OUT) :: inheritsched
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Returns current setting of the scheduling inheritance in thread attribute.

```
SUBROUTINE fpthrd_attr_getschedparam(attr, param, status)
```

```
TYPE (fpthrd_attr_t), INTENT(IN)  :: attr
```

```
    TYPE (fsched_param), INTENT(OUT) :: param
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Returns current setting of parameters used with scheduling in thread attribute.

```
SUBROUTINE fpthrd_attr_getschedpolicy(attr, policy, status)
```

```
TYPE (fpthrd_attr_t), INTENT(IN)  :: attr
```

```
    INTEGER, INTENT(OUT) :: policy
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Returns current setting of scheduling policy in thread attribute.

```
SUBROUTINE fpthrd_attr_getscope(attr, scope, status)
```

```
TYPE (fpthrd_attr_t), INTENT(IN)  :: attr
```

```
    INTEGER, INTENT(OUT) :: scope
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Returns current setting of scheduling scope in thread attribute.

```
SUBROUTINE fpthrd_attr_getstacksize(attr, stacksize, status)
```

```
TYPE (fpthrd_attr_t), INTENT(IN)  :: attr
```

```
    TYPE (fsize_t), INTENT(OUT) :: stacksize
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Returns the current setting of stack size in thread attribute.

```
SUBROUTINE fpthrd_attr_init(attr, status)
TYPE (fpthrd_attr_t), INTENT(OUT) :: attr
    INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Initializes thread attribute object to default settings.

```
SUBROUTINE fpthrd_attr_setdetachstate(attr, detachstate, status)
TYPE (fpthrd_attr_t), INTENT(INOUT) :: attr
    INTEGER, INTENT(IN) :: detachstate
    INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Sets detach state in thread attribute.

```
SUBROUTINE fpthrd_attr_setinheritsched(attr, inherit, status)
TYPE (fpthrd_attr_t), INTENT(INOUT) :: attr
    INTEGER, INTENT(IN) :: inherit
    INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Sets scheduling inheritance in thread attribute.

```
SUBROUTINE fpthrd_attr_setschedparam(attr, param, status)
TYPE (fpthrd_attr_t), INTENT(INOUT) :: attr
    TYPE (fsched_param), INTENT(IN) :: param
    INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Sets scheduling parameters in thread attribute.

```
SUBROUTINE fpthrd_attr_setschedpolicy(attr, policy, status)
TYPE (fpthrd_attr_t), INTENT(INOUT) :: attr
    INTEGER, INTENT(IN) :: policy
    INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Sets scheduling policy in thread attribute.

```
SUBROUTINE fpthrd_attr_setscope(attr, scope, status)
TYPE (fpthrd_attr_t), INTENT(INOUT) :: attr
    INTEGER, INTENT(IN) :: scope
    INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Sets scheduling scope in thread attribute.

```
SUBROUTINE fpthrd_attr_setstacksize(attr, stacksize, status)
TYPE (fpthrd_attr_t), INTENT(INOUT) :: attr
    TYPE (fsize_t), INTENT(IN) :: stacksize
    INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Sets stack size in thread attribute.

```
SUBROUTINE fpthrd_cancel(thread, status)
    TYPE (fpthrd_t), INTENT(IN) :: thread
    INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Sends cancellation signal to thread.

```
SUBROUTINE fpthrd_condattr_destroy(condattr, status)
```

```
TYPE (fpthrd_condattr_t), INTENT(OUT) :: condattr
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Destroys condition variable attribute object.

```
SUBROUTINE fpthrd_condattr_getpshared(condattr, pshared, status)
```

```
TYPE (fpthrd_condattr_t), INTENT(IN) :: condattr
```

```
INTEGER, INTENT(OUT) :: pshared
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Returns current process-shared setting in condition variable attribute.

```
SUBROUTINE fpthrd_condattr_init(condattr, status)
```

```
TYPE (fpthrd_condattr_t), INTENT(OUT) :: condattr
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Initializes condition variable object.

```
SUBROUTINE fpthrd_condattr_setpshared(condattr, pshared, status)
```

```
TYPE (fpthrd_condattr_t), INTENT(INOUT) :: condattr
```

```
INTEGER, INTENT(IN) :: pshared
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Sets process-shared setting in condition variable attribute.

```
SUBROUTINE fpthrd_cond_broadcast(cond, status)
```

```
TYPE (fpthrd_cond_t), INTENT(INOUT) :: cond
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Broadcast wakeup signal to all threads waiting on condition variable.

```
SUBROUTINE fpthrd_cond_destroy(cond, status)
```

```
TYPE (fpthrd_cond_t), INTENT(OUT) :: cond
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Destroy condition variable object.

```
SUBROUTINE fpthrd_cond_init(cond, condattr, status)
```

```
TYPE (fpthrd_cond_t), INTENT(OUT) :: cond
```

```
TYPE (fpthrd_condattr_t), INTENT(IN) :: condattr ! may be NULL
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Initialize condition variable object.

```
SUBROUTINE fpthrd_cond_signal(cond, status)
```

```
TYPE (fpthrd_cond_t), INTENT(INOUT) :: cond
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Signal at least one thread waiting on condition variable.

```
SUBROUTINE fpthrd_cond_timedwait(cond, mutex, timespec, status)
```

```
TYPE (fpthrd_cond_t), INTENT(INOUT) :: cond
```

```
TYPE (fpthrd_mutex_t), INTENT(INOUT) :: mutex
```

```
TYPE (ftimespec), INTENT(IN) :: timespec
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Wait on condition variable up to a specified length of time.

```
SUBROUTINE fpthrd_cond_wait(cond, mutex, status)
  TYPE (fpthrd_cond_t), INTENT(INOUT) :: cond
  TYPE (fpthrd_mutex_t), INTENT(INOUT) :: mutex
  INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Wait on condition variable.

```
SUBROUTINE fpthrd_create(thread, attr, routine, arg, status)
  TYPE (fpthrd_t), INTENT(OUT) :: thread
  TYPE (fpthrd_attr_t), INTENT(IN) :: attr ! may be NULL
  <type>, INTENT(IN) :: arg ! may be NULL
  INTEGER, INTENT(OUT) :: status
```

Create new thread executing subroutine `routine`. The subroutine `routine()` with single argument `arg` can be use-associated from a module or else be declared `EXTERNAL`.

```
SUBROUTINE fpthrd_detach(thread, status)
  TYPE (fpthrd_t), INTENT(IN) :: thread
  INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Set detach state of calling thread.

```
SUBROUTINE fpthrd_equal(thread1, thread2, flag)
  TYPE (fpthrd_t), INTENT(IN) :: thread1
  TYPE (fpthrd_t), INTENT(IN) :: thread2
  INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Compare one thread handle to another.

```
SUBROUTINE fpthrd_exit(value)
  INTEGER, INTENT(IN) :: value ! may be NULL
```

Terminate calling thread, returning value to any thread that joins terminated thread.

```
SUBROUTINE fpthrd_getschedparam(thread, policy, param, status)
  TYPE (fpthrd_t), INTENT(IN) :: thread
  INTEGER, INTENT(OUT) :: policy
  TYPE (fsched_param), INTENT(OUT) :: param
  INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Returns current scheduling parameters and policy of thread.

```
SUBROUTINE fpthrd_join(thread, value, status)
  TYPE (fpthrd_t), INTENT(IN) :: thread
  INTEGER, INTENT(OUT) :: value ! may be NULL
  INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Await termination of thread, return exit value of terminated thread.

```
SUBROUTINE fpthrd_mutexattr_destroy(mutexattr, status)
  TYPE (fpthrd_mutexattr_t), INTENT(OUT) :: mutexattr
  INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Destroy mutex attribute object.

```
SUBROUTINE fpthrd_mutexattr_getprioceiling(mutexattr, prioceiling, status)
TYPE (fpthrd_mutexattr_t), INTENT(IN)  :: mutexattr
      INTEGER, INTENT(OUT) :: prioceiling
      INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Returns current priority ceiling of mutex attribute.

```
SUBROUTINE fpthrd_mutexattr_getprotocol(mutexattr, protocol, status)
TYPE (fpthrd_mutexattr_t), INTENT(IN)  :: mutexattr
      INTEGER, INTENT(OUT) :: protocol
      INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Returns current protocol of mutex attribute.

```
SUBROUTINE fpthrd_mutexattr_getpshared(mutexattr, pshared, status)
TYPE (fpthrd_mutexattr_t), INTENT(IN)  :: mutexattr
      INTEGER, INTENT(OUT) :: pshared
      INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Returns current process-shared setting of mutex attribute.

```
SUBROUTINE fpthrd_mutexattr_init(mutexattr, status)
TYPE (fpthrd_mutexattr_t), INTENT(OUT) :: mutexattr
      INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Initializes mutex attribute object.

```
SUBROUTINE fpthrd_mutexattr_setprioceiling(mutexattr, prioceiling, status)
TYPE (fpthrd_mutexattr_t), INTENT(INOUT) :: mutexattr
      INTEGER, INTENT(IN)  :: prioceiling
      INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Sets priority ceiling of mutex attribute.

```
SUBROUTINE fpthrd_mutexattr_setprotocol(mutexattr, protocol, status)
TYPE (fpthrd_mutexattr_t), INTENT(INOUT) :: mutexattr
      INTEGER, INTENT(IN)  :: protocol
      INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Sets protocol of mutex attribute.

```
SUBROUTINE fpthrd_mutexattr_setpshared(mutexattr, pshared, status)
TYPE (fpthrd_mutexattr_t), INTENT(INOUT) :: mutexattr
      INTEGER, INTENT(IN)  :: pshared
      INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Sets process-shared setting of mutex attribute.

```
SUBROUTINE fpthrd_mutex_destroy(mutex, status)
TYPE (fpthrd_mutex_t), INTENT(OUT) :: mutex
      INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Destroys mutex.

```

SUBROUTINE fpthrd_mutex_init(mutex, mutexattr, status)
    TYPE (fpthrd_mutex_t), INTENT(OUT) :: mutex
TYPE (fpthrd_mutexattr_t), INTENT(IN)  :: mutexattr    ! may be NULL
    INTEGER, OPTIONAL, INTENT(OUT) :: status
Initializes mutex.

SUBROUTINE fpthrd_mutex_lock(mutex, status)
TYPE (fpthrd_mutex_t), INTENT(INOUT) :: mutex
    INTEGER, OPTIONAL, INTENT(OUT) :: status
Lock mutex if no other thread has control; else wait for lock to be released.

SUBROUTINE fpthrd_mutex_trylock(mutex, status)
TYPE (fpthrd_mutex_t), INTENT(INOUT) :: mutex
    INTEGER, OPTIONAL, INTENT(OUT) :: status
Lock mutex if no other thread has control; else return with exception.

SUBROUTINE fpthrd_mutex_unlock(mutex, status)
TYPE (fpthrd_mutex_t), INTENT(INOUT) :: mutex
    INTEGER, OPTIONAL, INTENT(OUT) :: status
Release mutex held.

SUBROUTINE fpthrd_once(once_block, init_routine, status)
TYPE (fpthrd_once_t), INTENT(INOUT) :: once_block
    INTEGER, OPTIONAL, INTENT(OUT) :: status
Ensures that init_routine will be executed by only one thread. The subroutine init_routine(),
with no arguments, can be use-associated from a module or else be declared EXTERNAL.

SUBROUTINE fpthrd_self(thread)
    TYPE (fpthrd_t), INTENT(OUT) :: thread
Return calling threads handle.

SUBROUTINE fpthrd_setcancelstate(state, oldstate, status)
    INTEGER, INTENT(IN) :: state
    INTEGER, INTENT(OUT) :: oldstate
    INTEGER, OPTIONAL, INTENT(OUT) :: status
Set cancel state of calling thread, return current cancel state.

SUBROUTINE fpthrd_setcanceltype(type, oldtype, status)
    INTEGER, INTENT(IN) :: type
    INTEGER, INTENT(OUT) :: oldtype
    INTEGER, OPTIONAL, INTENT(OUT) :: status
Set cancel type of calling thread, return current cancel type.

SUBROUTINE fpthrd_setschedparam(thread, policy, param, status)
    TYPE (fpthrd_t), INTENT(IN) :: thread
    INTEGER, INTENT(IN) :: policy
TYPE (fsched_param), INTENT(IN) :: param
    INTEGER, OPTIONAL, INTENT(OUT) :: status
Set scheduling parameters and policy of thread.

SUBROUTINE fpthrd_testcancel()
Accepts any pending cancellation signal.

```

APPENDIX B – Additional Fortran Routines

The details of additional support and utility routines within FPTHREAD are given in this appendix. A routine to get the timespec structure is not included because the seconds component of the structure is given in absolute time units, based on an unspecified previous time. Only the first routine listed here requires a **status** parameter since it is a defined Pthreads function that returns an exception code. The other routines do not need this parameter or do not correspond to any Pthreads function.

```
SUBROUTINE fpthrd_setconcurrency(new_level, status)
```

```
    INTEGER, INTENT(IN)  :: new_level
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: status
```

Give runtime system a hint about the number of threads to be executing concurrently.

```
SUBROUTINE fpthrd_getconcurrency(level)
```

```
    INTEGER, INTENT(OUT) :: level
```

Return current concurrency level; return 0 if level is being maintained automatically.

```
SUBROUTINE fpthrd_data_exchange()
```

Initialize Fortran Pthreads API and exchange parameter values.

```
SUBROUTINE fpthrd_set_fsize(size, fsize)
```

```
    INTEGER, INTENT(IN)  :: size
```

```
    TYPE (fsize_t), INTENT(OUT) :: fsize
```

Translate the fsize derived type to the corresponding value of size.

```
SUBROUTINE fpthrd_get_fsize(size, fsize)
```

```
    INTEGER, INTENT(OUT) :: size
```

```
    TYPE (fsize_t), INTENT(IN)  :: fsize
```

Retrieve the current size value within the fsize_t derived type.

```
SUBROUTINE fpthrd_set_fsched_param(schedule_value, sched)
```

```
    INTEGER, INTENT(IN)  :: schedule_value
```

```
TYPE (fsched_param), INTENT(INOUT) :: sched
```

Set the fsched_param derived type with the value of the given schedule_value constant.

```
SUBROUTINE fpthrd_get_fsched_param(schedule_value, sched)
```

```
    INTEGER, INTENT(OUT) :: schedule_value
```

```
TYPE (fsched_param), INTENT(IN)  :: sched
```

Retrieve the current value of the fsched_param derived type.

```
SUBROUTINE fpthrd_set_ftimespec(change_sec, change_nanosec, t)
```

```
    INTEGER, INTENT(IN)  :: change_sec
```

```
    INTEGER, INTENT(IN)  :: change_nanosec
```

```
    TYPE (ftimespec), INTENT(OUT) :: t
```

Set the updated absolute units of seconds and nanoseconds in the ftimespec derived type.